METHOD AND APPARATUS FOR SWITCHING BETWEEN MULTIPLE IMPLEMENTATIONS OF A ROUTINE

Cary A. Coutant
Carol L. Thompson

FIELD OF THE INVENTION

The present invention generally relates to management of binary program code for different implementations of a processor architecture, and more particularly to switching between multiple implementations of a routine that are associated with the implementations of a processor architecture.

BACKGROUND

It is common to build multiple implementations of a basic processor architecture for the purpose of providing various performance and pricing options. For example, a processor architecture that supports a particular instruction set may have a first-level cache in one implementation, and another implementation may have first and second level caches.

Some system-provided routines and other library routines are written and compiled to exploit the performance characteristics of a particular implementation of a processor. For example, a memory-to-memory copy routine may be written and compiled differently from one implementation to another. While the binary code will execute correctly on any implementation, there may be a negative impact on performance when the binary code is executed on an implementation other than the target implementation.

A software developer can either develop separate binary libraries for the different implementations, develop a single binary for all implementations, or develop several versions of selected routines along with a run-time switch for selecting between the different versions. Developing separate binary libraries is technically straightforward. However, there is a cost associated with managing and distributing separate binary libraries. If only a single binary library is provided, users may not receive the full performance benefit of a particular implementation. While run-time switches would appear to provide a reasonable tradeoff between the management of multiple binary libraries and performance degradation, the run-time switch introduces overhead when a library call is made to determine the implementation on which the code is executing.

A method and apparatus that address the aforementioned problems, as well as other related problems, are therefore desirable.

SUMMARY OF THE INVENTION

In various embodiments, a method and apparatus are provided for switching between multiple implementations of a routine. In one embodiment, different versions of a library routine are programmed to exploit different features of different computer systems. The different versions are available in a single library, and an application program need not differentiate between the different implementations in using the routine. Using hardware characteristics that are associated with the different versions and hardware characteristics of the computer system on which the application is to be executed, references to the routine are resolved when the application and library are loaded. Thus, execution of the application is not burdened with runtime resolution of references to the routine.

It will be appreciated that various other embodiments are set forth in the Detailed Description and Claims which follow.

BRIEF DESCRIPTION OF THE DRAWINGS

Various aspects and advantages of the invention will become apparent upon review of the following detailed description and upon reference to the drawings in which:

FIG. 1 is a flowchart of a process for generating multiple binary implementations of a routine for different implementations of a particular processor architecture;

FIG. 2 is a block diagram illustrating a symbol table in relationship to a shared library of object code modules; and

FIG. 3 is a flowchart of a process for loading a library routine in accordance with one embodiment of the invention.

DETAILED DESCRIPTION

In various embodiments, the invention provides a technique for switching between multiple implementations of a library routine that are available in a library of routines. Each implementation of a routine has an associated set of hardware characteristics that indicate the hardware on which the implementation is intended to execute. The hardware characteristics may include, for example, the processor clock speed or a model number, cache configuration, latency of selected hardware operations (load and store, for example), and the availability of certain extensions to the instruction set. When a routine having multiple implementations is loaded, the reference is resolved to the appropriate implementation using the associated hardware characteristics and the hardware characteristics of the host system. Additional hardware characteristics that may be used for different implementations of routines include, for example, bypass characteristics, branch prediction behavior, pre-fetching capability, information describing stall conditions, branch penalties, size and associativity of processor data structures (not just cache, but branch prediction and ALAT-like structures as well), queue sizes for out-of-order or decoupled processors, and the number of processors in a multi-processor system.

FIG. 1 is a flowchart of a process for generating multiple binary implementations of a routine for different implementations of a particular processor architecture, in accordance with one embodiment of the invention. The process generally entails for each implementation of a routine, generating object code modules for the different implementations and adding entries in a symbol table for the different object code modules. An entry in the symbol table for a routine having multiple implementations has an associated set of hardware characteristics. The hardware characteristics are those of the platform on which the associated object code module is intended to execute.

At step 102, the first (or next, depending on the iteration) implementation of the routine is obtained for processing, and at step 104 the hardware characteristics associated with the routine are obtained.

The symbol table is updated at step 106. The symbol table includes names of routines and references to the associated object code modules. For routines having multiple implementations, hardware characteristics are also associated with the routine name in the symbol table. When an application is loaded and bound to the shared library that contains the multiple binary implementations of a routine, the system dynamic loader selects the appropriate implementation from the symbol table based on the hardware characteristics of the host system.

In one embodiment, the hardware characteristics that are to be associated with a routine are provided by the developer in a configuration file that is read by the linker at the time the shared library is being built. In this embodiment, the programmer will have coded different versions of the routine and used different names for the different versions. The information in the configuration file associates the names of the different versions with sets of hardware characteristics and with a generic name.

In another embodiment, the compiler could be adapted to generate multiple object code modules from a source code module. For example, in response to a command-line option, the compiler automatically generates hardware-specialized implementations, generates unique symbol names for the specialized implementations, and generate the mapping information. The mapping information associates the generic routine name with the specialized implementations, and the specialized implementations with sets of hardware characteristics. The mapping information may be stored either in the object file or in a separate configuration file that is used by the linker, for example.

At step 108, the object code module is created for the routine, and the object code module is added to the shared library at step 110. Each routine in the shared library is assigned a unique name.

At step 112, the entry in the symbol table (step 106) is updated to reference the associated object code module in the object code library. Decision step 114 tests whether there are additional implementations to process. If so, control is returned to step 102. Otherwise, the process for generating the multiple implementations is complete.

FIG. 2 is a block diagram illustrating a symbol table in relationship to a shared library of object code modules. The routines in shared library 152 are object code modules that are associated with and referenced by the entries in symbol table 154.

Routines 1 - (n+1) are illustrated in shared library 152. Routines 1 - n have single implementations, and two implementations are illustrated for example routine (n + 1). The first implementation of routine (n + 1) is named routine (n + 1), and the second implementation of routine (n + 1) is named routine (n + 1)'.

Symbol table 154 includes entries for each routine and implementation. Routines 1 - n, having only single implementations, include only the routine name and a reference to the corresponding object code module in shared library 152. Routine (n + 1) has two implementations, and the entries associated therewith include respective sets of hardware characteristics that describe, for example, the processor for which the implementations were developed. The entry having hardware characteristics set 1 references routine (n + 1) code in shared library 152, and the entry having hardware characteristics set 2 references routine (n + 1)' code in the shared library.

When an application is loaded and bound to shared library 152, the system dynamic loader selects the appropriate binary implementation from the symbol table based on the hardware characteristics of the host processor. Thus, the reference to the appropriate binary implementation is resolved when the program using the shared library is loaded. Alternatively, some environments may load the shared library after a program begins execution, and in this environment the references are resolved when the shared library is loaded. In either environment, the references are resolved at load time versus runtime. The resolution of the references to routines in the symbol table results in code references to the addresses of the binary implementations in the shared library 152.

By selecting the appropriate object code routine once when the routine is first referenced instead of resolving the reference each time the routine is referenced at run-time, the overhead for the switch occurs in the compiler and loader, thereby eliminating issues with respect to run-time performance and switching to an appropriate implementation of a routine.

FIG. 3 is a flowchart of a process for loading a library routine in accordance with one embodiment of the invention. At step 302, the hardware characteristics are obtained from a system configuration file, for example. In another embodiment, the characteristics may be obtained, for example, from hardware identification registers or from firmware.

At step 304, the loader obtains the name of the routine to be loaded. For example, an application program may reference a particular shared library routine, and the loader uses the program-specified routine name to locate the proper object code module in the shared library.

The routine name and hardware characteristics are used at step 306 to match an entry in symbol table 154. Using the reference in the matching entry, step 208 loads the referenced object that is associated with the hardware characteristics. Forward from the time that a routine is referenced and the proper object code module is identified and loaded, no further matching of hardware characteristics is required on subsequent references to the routine.

The present invention is believed to be applicable to a variety of systems that switch between multiple implementations of a routine based on hardware characteristics. Other aspects and embodiments of the present invention will be apparent to those skilled in the art from consideration of the specification and practice of the invention disclosed herein. It is intended that the specification and illustrated embodiments be considered as examples only, with a true scope and spirit of the invention being indicated by the following claims.